APPLICATION FOR UNITED STATES PATENT

in the name of

Soeren Laursen, Morten Stribaek, and Hartvig Ekner

for

Binary Polynomial Multiplier

Fish & Richardson P.C.
601 Thirteenth Street, NW
Washington, DC 20005
Tel.: (202) 783-5070
Fax: (202) 783-2331

ATTORNEY DOCKET:
12135-006001

Binary Polynomial Multiplier

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is related to the following co-pending applications, each of which is being filed concurrently with this application and is incorporated by reference: (1) U.S. Application No. _____, titled "Partial Bitwise Permutations"; (2) U.S. Application No. _____, titled "Configurable Instruction Sequence Generation"; (3) U.S. Application No. _____, titled "Polynomial Arithmetic Operations"; and (4) U.S. Application No. _____, titled "Extended Precision Accumulator".

TECHNICAL FIELD

This invention relates to a microprocessor multiplier, and more particularly to a microcomputer multiplier supporting binary polynomial multiplication.

BACKGROUND

Reduced instruction set computer (RISC) architectures were developed as industry trends tended towards larger, more complex instruction sets. By simplifying instruction set designs, RISC architectures make it easier to use techniques such as pipelining and caching, thus increasing system performance.

RISC architectures usually have fixed-length instructions (e.g., 16-bit, 32-bit, or 64-bit), with few variations in instruction format. Each instruction in an instruction set architecture (ISA) may have the source registers always in the same location. For example, a 32-bit ISA may always have source registers specified by bits 16-20 and 21-25. This allows the specified registers to be fetched for every instruction without requiring any complex instruction decoding.

SUMMARY

Cyrptographic systems ("cryptosystems") are increasingly used to secure transactions, to encrypt communications, to authenticate users, and to protect information. Many private-key cryptosystems, such as the Digital Encryption Standard (DES), are relatively simple computationally and frequently reducible to hardware solutions performing

sequences of XORs, rotations, and permutations on blocks of data. Public-key cryptosystems, on the other hand, may be mathematically more subtle and computationally more difficult than private-key systems.

While different public-key cryptography schemes have different bases in mathematics, they tend to have a common need for integer computation across very large ranges of values, on the order of 1024 bits. This extended precision arithmetic is often modular (i.e., operations are performed modulo a value range), and in some cases polynomial instead of twos-complement. For example, RSA public-key cryptosystems use extended-precision modular exponentiation to encrypt and decrypt information and elliptic curve cryptosystems use extended-precision modular polynomial multiplication.

Public-key cryptosystems have been used extensively for user authentication and secure key exchange, while private-key cryptography has been used extensively to encrypt communication channels. As the use of public-key cryptosystems increases, it becomes desirable to increase the performance of extended-precision modular arithmetic calculations.

In one general aspect, a multiply/divide unit is provided. The multiply/divide unit includes at least one input register for storing one or more input operands, an arithmetic multiplier, a binary polynomial multiplier, and at least one result register.

Some implementations may provide an arithmetic multiplier including a multiplier array, such as, for example, a Wallace tree multiplier array. The multiplier array may include carry-save adders arranged in a tree structure and also may include carry-propagate adders.

Implementations may include Booth recoding logic to simplify computation. The multiplier/divide unit may perform 32-bit by 16-bit multiplications in two clock cycles and 32-bit by 32-bit multiplications in three clock cycles. Additional implementations may include a binary polynomial multiplier array.

The details of one or more implementations are set forth in the accompanying drawings and the description below. Other features and advantages will be apparent from the description and drawings, and from the claims.

DESCRIPTION OF DRAWINGS

FIG. 1 is a block diagram of an exemplary five-stage pipeline that may be used in a RISC architecture.

FIG. 2 is a block diagram of a processor core including an execution unit and a multiply/divide unit.

FIG. 3 is a diagram of data paths in an implementation of a multiply/divide unit supporting binary polynomial arithmetic.

FIG. 4 is a block diagram of multiplier arrays supporting arithmetic and binary polynomial multiplication in one implementation.

FIG. 5 is a block diagram of an integer multiplier array that may be used in the implementation shown in FIG. 4.

FIG. 6 is a block diagram of a binary polynomial multiplier array that may be used in the implementation shown in FIG. 4.

FIG. 7A is a timing diagram showing the operation of 32-bit by 16-bit multiplies in one implementation.

FIG. 7B is a timing diagram showing the operation of 32-bit by 32-bit multiplies in one implementation.

FIG. 7C is a timing diagram showing the operation of divisions in one implementation.

FIG. 8 is finite state machine implementing steps for performing multiply instructions.

FIG. 9 is a finite state machine implementing steps for performing division instructions.

DETAILED DESCRIPTION

Many public-key cryptosystems use extended-precision modular arithmetic to encrypt and decrypt data. For example, many elliptic curve (EC) cryptosystems extensively use polynomial multiplication and addition to encrypt and decrypt data. Performance of elliptic curve cryptosystems may be enhanced by modifying a programmable CPU multiplier to be responsive to newly defined instructions dedicated to polynomial operations.

When using elliptic curves defined over $GF(2^{163})$ (as recommended by the IEEE 1363-2000 standard), the main operation needed is multiplication over the field $GF(2^{163})$. Each of the 2^{163} elements can be represented as a polynomial of degree at most 163 with coefficients equal to 0 or 1. In this representation, two elements may be added using a simple

bitwise XOR, and two polynomials, $a(X)$ and $b(X)$, may be multiplied by computing $a(X)b(X)$ mod $P(X)$, where the product $a(X)b(X)$ is a 326-degree polynomial, and $P(X)$ is an irreducible polynomial as specified by the IEEE 1363-2000 standard.

Polynomial multiplication has the same form as modular multiplication, ab mod p, over the integers, except that: (1) regular addition is replaced by an XOR; and (2) regular 32-bit multiplication is replaced by a 32-bit carry-free multiplication. Therefore, polynomial modular multiplication may be performed using shifts and XORs instead of shifts and adds.

It is desirable to provide a multiply/divide unit that supports fast polynomial multiplication and various other operations to increase the performance of cryptographic and other systems.

Referring to FIG. 1, an exemplary microprocessor architecture that may be used to implement polynomial multiplication includes a five-stage pipeline in which each instruction is executed in, for example, five clock cycles. The execution of each instruction is divided into five stages: instruction fetch (IF) stage 1001, register read (RD) stage 1002, arithmetic/logic unit (ALU) stage 1003, memory (MEM) stage 1004, and write back (WB) stage 1005. In the IF stage 1001, a specified instruction is fetched from an instruction storage. A portion of the fetched instruction is used to specify source registers that may be used in executing the instruction. In the read registers (RD) stage 1002, the system fetches the contents of the specified source registers. These fetched values may be used to perform arithmetic or logical operations in the ALU stage 1003. In the MEM stage 1004, an executing instruction may read/write memory in a data storage. Finally, in the WB stage 1005, values obtained by the execution of the instruction may be written back to a register.

Because the results of some operations, such as floating point calculations and integer multiply/divide, may be not be available in a single clock cycle, instructions to perform these operations merely initiate execution. After sufficient clock cycles have passed, another instruction may be used to retrieve a result. For example, when an integer multiply instruction takes five clock cycles, one instruction may initiate the multiplication calculation, and another instruction may load the results of the multiplication into a register after the multiplication has completed. If a multiplication has not completed by the time a result is requested, the pipeline may stall until the result is available.

Referring to FIG. 2, an exemplary RISC architecture is provided by way of example. The processor core 2000 includes: an execution unit 2010, a multiply/divide unit (MDU) 2020, a system control coprocessor (CP0) 2030, a memory management unit 2040, a cache controller 2050, and a bus interface unit (BIU) 2060. MDU 2020 is a combined multiply/divide unit in one implementation. Other implementations may provide separate multiply and divide units.

Execution unit 2010 is the primary mechanism for executing instructions within processor core 2000. Execution unit 2010 includes a register file 2011 and an arithmetic logic unit (ALU) 2012. In one implementation, the register file 2011 includes 32 32-bit general-purpose registers that may be used, for example, in scalar integer operations and address calculations. The register file 2011, which includes two read ports and one write port, may be fully bypassed to minimize operation latency in the pipeline. ALU 2012 supports both logical and arithmetic operations, such as addition, subtraction, and shifting.

The MDU 2020 includes three registers (ACX 2021, HI 2022, and LO 2023) that may be used for various operations. In accordance with one implementation, these three registers may be used together to hold up to a 72-bit value. In one implementation, LO register 2023 and HI register 2022 are each 32 bits wide and function as dedicated output registers of MDU 2020. In one implementation, ACX register 2021 provides 8 bits of additional integer precision beyond those provided by the HI/LO register pair. The precise number of bits is implementation dependent, with the preferred minimum size being 2 bits. For processors with 32 bit data paths, the preferred maximum size of the ACX register is 32 bits. In contrast, for processors with 64 bit data paths, the preferred maximum size of the ACX register is 64 bits. Hence, in a processor with 32-bit wide HI and LO registers, the combination of ACX/HI/LO can hold a concatenated value having more than 64 bits.

The MDU 2020 may be used to perform various operations including some or all of the following instructions, which are described below: DIV, DIVU, MADD, MADDU, MFHI, MFLO, MSUB, MSUBU, MTHI, MTLO, MUL, MULT, MULTU, MFLHXU, MTLHX, MADDP, MULTP, and PPERM.

The instructions MUL, MULT, and MULTU may be used to multiply two 32-bit numbers together. The result is stored in a specified register for MUL, and in the HI/LO registers for MULT and MULTU. For example, "MUL $7, $6, $5" multiplies the contents of

registers $6 and $5 together and stores the result in register $7. The instruction "MULT $6, $5" multiplies the contents of registers $6 and $5 together and stores the result in the HI/LO registers. The MULTU instruction performs the same operation as MULT; however, MULTU applies to unsigned operands and MULT applies to signed operands. Additionally, the MULTU instruction clears the ACX register to all zeros.

The instructions DIV and DIVU perform division operations and store the results in the accumulator registers. For example, "DIV $6, $5" divides the contents of register $6 by the contents of register $5 and stores the quotient in the LO register and the remainder in the HI register. The DIVU instruction performs the analogous operation on unsigned operands.

The instructions MSUB, MSUBU, MADD, and MADDU may be used to multiply the contents of two registers and then add or subtract the contents of the ACX/HI/LO registers. For example, "MSUB $6, $5" multiplies the contents of registers $6 and $5 together, subtracts the result from the contents of the ACX/HI/LO registers, and then stores the result in the ACX/HI/LO registers. The MADD instruction similarly multiplies the contents of two registers, adds the result to the ACX/HI/LO registers, and stores the result in the ACX/HI/LO registers. The MSUBU and MADDU perform the analogous operations to unsigned operands. In some implementations, the value written to the ACX register is only defined in a subset of these operations.

The following instructions are used to move data between the ACX/HI/LO registers and general purpose registers: MFHI, MFLO, MTHI, MTLO, MFLHXU, and MTLHX. The first instruction, MFHI, loads the contents of the HI register into a general purpose register. For example, "MFHI $5" loads the contents of the HI register into register $5. Similarly, MFLO loads the contents of the LO register into a general purpose register. Conversely, the instructions MTHI and MTLO are used to load the contents of a general purpose register into the HI or LO registers. For example, "MTHI $5" loads the contents of register $5 into the HI register.

In one implementation, the content of the ACX register is not directly accessible. To indirectly access the ACX register, the values stored in the ACX/HI/LO registers may be shifted to the left or right. For example, "MFLHXU $5" shifts contents of the ACX, HI, and LO registers to the right by one register position and loads the contents of the LO register into register $5. Thus, after performing the operation, the ACX register is zero, the HI

register contains the previous contents of the ACX register, the LO register contains the previous contents of the HI register, and the $5 register contains the previous contents of the LO register. If the ACX register is narrower than the HI register, the contents of the ACX register may be zero-extended appropriately before loading the HI register. The MTLHX performs the inverse operation. For example, "MTLHX $5" loads the ACX register with the low-order 8 bits of the previous contents of the HI register, loads the HI register with the previous contents of the LO register, and loads the LO register with the contents of the $5 register.

The PPERM operation performs permutations as specified in a register, storing the result in the ACX/HI/LO registers. For example, "PPERM $5, $6" causes the ACX/HI/LO registers to be shifted 6-bits to the left. Then, low-order six bits are selected from register $5 as specified by register $6. In particular, the 32-bit contents of register $6 are used to select which bits of register $5 will be used to fill the low-order bits of the ACX/HI/LO registers. Since there are 32 bits in register $5, 5 bits are needed to specify a specific one of the 32 bits. For example, "01101" is binary for the number 13. Thus, these five bits may specify bit 13. Similarly, "00000" is binary for 0 and "11111" is binary for 31. Thus, any one of the 32 bits may be specified using a 5-bit specifier, and 6 bits may be specified using 30 bits (i.e., 6 5-bit specifiers).

Register $6 may specify the bits of register $5 used to fill the low-order bits of ACX/HI/LO as follows: bits 0-4 are used to specify the source of bit 0, bits 5-9 are used to specify bit 1, bits 10-14 are used to specify bit 2, bits 15-19 are used to specify bit 3, bits 20-24 are used to specify bit 4, and bits 25-29 are used to specify bit 5. The remaining bits, 30-31, may be unused. Thus, the instruction is performed using the specifiers as described to fill the lowest 6 bits of the LO register with the specified bits from register $5.

Finally, MULTP may be used to perform binary polynomial multiplication and MADDP may be used to perform binary polynomial multiplication with the result added to the ACX/HI/LO registers. These operations are analogous to MULT and MADD, but operate on binary polynomial operands.

The polynomial operands of MULTP and MADDP are encoded in 32-bit registers with each bit representing a polynomial coefficient. For example, the polynomial "$x^4 + x + 1$" would be encoded as "10011" because the coefficients of x^3 and x^2 are "0"

and the remaining coefficients are "1". The MULTP instruction performs binary polynomial multiplication on two operands. For example,

$$(x^4 + x + 1)(x + 1) = x^5 + x^4 + x^2 + 2x + 1.$$

Reducing the polynomial modulo two, yields $x^5 + x^4 + x^2 + 1$. If the polynomials are encoded in the binary representation above, the same multiplication may be expressed as $(10011)(11) = 110101$.

The MADDP instruction performs multiplication just as MULTP, and then adds the result to the ACX/HI/LO registers. Polynomial addition may be performed using a bitwise XOR. For example, the binary polynomial addition $(x^4 + x + 1) + (x + 1)$ yields $x^4 + 2x + 2$. Reducing the coefficients modulo 2 yields x^4 which may be expressed as "10000".

Referring to FIG. 3, MDU 2020 receives two 32-bit operands, RS and RT. Using these operands, MDU 2020 performs a requested operation and stores a result in registers ACX 2021, HI 2022, and LO 2022. Major data paths that may be used to perform these operations are shown in FIG. 3. The RShold register 3010 and the RThold register 3012 are used to hold the RS and RT operands. Multiplexers 3020, 3022, and 3024 are used to select whether to use the RS and RT operands directly or to use the values stored in RShold 3010 and RThold 3012. Additionally, multiplexer 3022 may be used to select between the low-order and high-order bits of RT.

The RThold register 3012 is connected to multiplexer 3022. Multiplexer 3022 produces a 16-bit result by selecting the high-order bits of RThold 3012, the low-order bits of RThold 3012, the high-order bits of the RT operand, or the low-order bits of the RT operand. The output from multiplexer 3022 is processed by Booth recoder 3040 and stored in register RTB 3042. The output of register RTB 3042 becomes the input SEL 3034 to array unit 3030.

Array unit 3030 is used to perform arithmetic and binary polynomial multiplication as described below with reference to FIG. 4. Array unit 3030 takes as inputs ACC1 3031, ACC2 3032, M 3033, SEL 3034, and RThold 3012. Inputs ACC1 3031 and ACC2 3032 are accumulated results used for operations that perform a multiplication and add or subtract the resulting value from an accumulated result. The inputs SEL 3034 (determined by register RTB 3042) and M 3033 (determined by register RShold 3011) form the operands for arithmetic operations. The inputs RThold 3012 (or the high-order or low-order bits of

RThold 3012) and M 3033 (determined by RShold 3011) form operands for polynomial operations and permutations. Combinations of these inputs are used to perform various calculations as described in detail below.

Array unit 3030 also includes two outputs, ResultC 3035 and ResultS 3036. In performing arithmetic operations, carry-save adders (CSAs) may be used to build a multiplication array. Carry-save adders calculate sums and carries separately to produce two outputs. Thus, ResultC 3035 and ResultS 3036 represent the carry and the sum outputs of a CSA multiplier array. In one implementation, ACC1 3031, ACC2 3032, ResultC 3035, and ResultS 3036 are each 72 bits long (to support a 72-bit extended-precision accumulator with an 8-bit ACX register, a 32-bit HI register, and a 32-bit LO register) and the remaining inputs are at most 32 bits long. Inputs ACC1 3031 and ACC2 3032 may be selected using multiplexers 3037 and 3038.

Multiplexers 3050 and 3052 are used to select values as inputs to registers CPAA 3054 and CPAB 3056. For example, multiplexer 3050 may be used to select ResultC 3035, the output of CPA 3058, operand RS, or RShold 3010, and multiplexer 3052 may be used to select ResultS 3036, the value 0, operand RT, or RThold 3012. These registers store the inputs to carry-propagate adder (CPA) 3058. CPA 3058 may be used to complete multiplication operations (multiplies) and to perform iterative division operations (divides) as discussed below.

Register RDM 3060 stores the result of CPA 3058. Finally, multiplexer 3070 selects which values form the result to be loaded into registers ACX, HI, and LO. Multiplexer 3070 may be used to select the ACX/HI/LO registers, RDM 3060, or the result of CPA 3058. Multiplexer 3072 may be used to instead load various permutations of the result selected by multipexer 3070. Multiplexer 3072 is used to perform various shifts of the computed result and concatenations with RShold 3010 by selecting from the following values (forming 72-bit values when concatenated): (1) the 72-bit output of multiplexer 3070; (2) the 8 high-order bits of multiplexer 3070, the contents of RShold 3010, and the 32 low-order bits of multiplexer 3070; (3) the 40 high-order bits of multiplexer 3070 and the contents of RShold 3010; (4) the 40 low-order bits of multiplexer 3070 and the contents of RShold 3010; and (5) the 40 high-order bits of multiplexer 3070 (with 32 leading zeros).

Some operations cause the values stored in the result registers ACX, HI, and LO to be modified as part of a read operation. For this reason, a separate result register 3080 may be provided to store the high-order and low-order result without the accumulator ACX.

The data path described below includes six major parts: (1) input registering and selection; (2) Booth recoding; (3) multiplier arrays and permutation logic; (4) a carry-propagate adder; (5) result registering and selection; and (6) a separate 32-bit output register for presenting results.

Input registering and selection is performed using the RShold and RThold registers to hold the RS and RT operands. Multiplexers select whether to use these operands directly or to use the registered versions. Booth recoding is performed on half of the RT operand at a time to provide inputs to the multiplier arrays and permutation logic. Hennessy and Patterson describe Booth recoding in Appendix A of "Computer Architecture: A Quantitative Approach," Morgan Kaufmann Publishers, Inc. (1996), which is hereby incorporated by reference in its entirety for all purposes.

One array of array unit 3030 performs arithmetic multiplication and one array of array unit 3030 performs binary polynomial multiplication. In one implementation, both arrays are 32 bits by 16 bits (32x16) and are used once or twice depending on the size of the RT operand (i.e., an appropriate array is used once when the value of RT can be held in the low-order 16 bits and twice otherwise). The CPA may be used to complete multiplies and perform iterative divides. Other implementations may include faster mechanisms for performing divides.

The arithmetic multiplication array may be implemented using any of the techniques described by Hennessy and Patterson in "Computer Architecture: A Quantitative Approach," Morgan Kaufmann Publishers, Inc. (1996), which is incorporated by reference in its entirety for all purposes. For example, Appendix A of Hennessy and Patterson describes several ways to speed up arithmetic multipliers. Any of the described techniques may be used as a basis for the polynomial multiplication extensions described below.

Referring to FIG. 4, array unit 3030 includes two parallel multipliers (Marray 4100 and MParray 4200) and permutation logic 4300. The first array, Marray 4100, performs arithmetic multiplication as described below with reference to FIG. 5. Marray 4100 uses the following inputs as described above: ACC1 3031, ACC2 3032, M 3033, and SEL 3034. The

outputs include ResultC 3035 and ResultS 3036. The second array, MParray 4200, performs binary polynomial multiplication as described below with reference to FIG. 6. MParray 4200 uses the following inputs as described above: the low-order bits of RThold 3012 or the high-order bits of RThold 3012; RShold 3010; and ACC1 3031. The output of MParray 4200 is ResultC 3036. Finally, permutation logic 4300 is used to perform various permutations on the low-order bits of RShold 3010 based on the value stored in RThold 3012.

Referring to FIG. 5, Marray 4100 is a 32-bit by 16-bit Wallace tree multiplier array that has been modified to support the addition of two 72-bit wide operands ACC1 and ACC2. The ACC1 and ACC2 operands hold a carry-save representation of a 72-bit value. Because additions are already performed to carry out multiplications (i.e., by the carry-save adders (CSAs)), an additional adder may be included to add ACC1 and ACC2 to intermediate results of multiplications. Marray 4100 generates a 72-bit wide result in a carry-save representation. Since 32x16 bits are processed per cycle, two passes through the array are required for 32x32 bit multiplies.

Array Marray 4100 is implemented as a Wallace tree built from arrays of carry-save adders. The width of these arrays may vary. Because the accumulate value from the previous array pass is input late into the array, the accumulate value does not need to come directly from a register. Booth recoding is performed using the method of overlapping triplets to more efficiently process multiplications. The output of Booth recoding tells whether to add operand M multiplied by -2, -1, 0, 1, or 2 for each power of 4. The multiplexers on the top-level CSA inputs are used to select the corresponding multiple of M.

Array Marray 4100 accumulates eight products from the Booth recoding plus one special partial product. The latter may be used for 32-bit unsigned calculations using the "0" and "1x" choices from the multiplexers. Within the Wallace tree, operands may be sign-extended to properly accumulate 2's complement results.

Referring to FIG. 6, binary polynomial-based multiplication operations are processed similarly to corresponding unsigned arithmetic operations. In one implementation, MParray 4200 is a 32x16 bit array that also performs an addition using exclusive-or (XOR) on an operand, for example, ACC1. As with Marray 4100, 32x16 bits are processed per cycle and two passes through the array may be used for 32x32 multiplies. In the first cycle, ACC1 is

zero (for a MULTP operation) or the previous result (for a MADDP operation). In a second cycle, ACC1 is the high order bits of the output from the first cycle.

MParray 4200 multiplies two operands (e.g., OpA and OpB) using an array with each row formed by taking the AND of OpA and a bit of OpB. For example, the first row is the logical AND of OpA and bit 0 of OpB. Row two is the logical AND of OpA and bit 1 of OpB. The result of each successive row is shifted one bit to the left. The final result is formed by taking the exclusive-or (XOR) of each column. Because bitwise XOR may be used to perform addition in binary polynomial arithmetic, an accumulator row may be added to array MParray 4200 to support instructions such as MADDP.

In one implementation, permutation logic 4300 is used to support the PPERM instruction described above. Permutation logic 4300 consists of 6 single bit 32:1 selectors that may be used to select any 6 of the 32 bits of RShold 3010 based on the value of RThold 3012.

For example, permutation logic 4300 may be used to execute the instruction "PPERM $5, $6". Permutation logic 4300 uses 6 5-bit selectors determined by RThold 3012 to identify which bits to include as output from RShold 3010. For example, if register $5 contains the low-order bits "010101", then the selector "00010" would choose bit 2 (i.e., the third bit from the right) containing "1". If RThold 3012 contains the low-order bits "0001000011", then bit 2 (containing a "1") and bit 3 (containing a "0") will be selected yielding "10". Using this method, permutation logic 4300 may select bits from RShold 3010 to generate 6 bits based on RThold 3012. The resulting 6 bits concatenated to the 66 low-order bits of ACC1 to form the result. This effectively shifts the 66 low-order bits of ACC1 six bits to the left and replaces the 6 low-order bits with the output of the permutation logic 4300.

Three multiplexers shown in FIG. 4 are used to select either zero or the sum output of Marray 4100 to form ResultS 3036; and the output of Marray 4100, MParray 4200, or permutation logic 4300 to form ResultC 3035.

Referring again to FIG. 1, MDU 2020 starts a computation in the first cycle of the ALU stage of the pipeline 1003. If the calculations complete before the instruction has moved past the MEM stage 1004 in the pipeline, then the result is held at that point. If the operation completes when the instruction has been moved past the MEM 1004 in the

pipeline, then the instruction has been committed and the results are written directly to the ACX/HI/LO registers.

The MDU 2020 is decoupled from the environment pipeline; it does not stall with the environment. That is to say the MDU 2020 will continue its computation during pipeline stalls. In this way, multi-cycle MDU operations may be partially masked by system stalls and/or other, non-MDU instructions.

Referring to FIG. 7A, for 32x16 bit multiplies, the pipeline flow through MDU 2020 is as follows. The first cycle may be used for Booth recoding. The second cycle is where the array is run and the third cycle is where the CPA 3058 completes the computation. The result of a 32x16 bit multiply is available soon enough for a following MFxx instruction without stalling the pipeline. A 32x16 MUL, which returns the result directly to a general purpose register (GPR) may stall for one cycle.

Referring to FIG. 7B, for 32x32 bit multiplies, the array is used twice, which adds one extra clock cycle to the 32x16 bit multiplications. As the first array pass is completing for the first portion of operand RT, Booth recoding is performed on the second portion of the operand. Thus, the Booth recoded portion of RT is available to begin the second pass through the array immediately after the first pass is complete. The multiplication result is then calculated using CPA 3058.

Referring to FIG. 7C, a simple non-restoring division algorithm may be used for positive operands. The first cycle is used to negate RS, if needed. Following that, a varying number of iterative add/subtracts are performed. The actual number is based on the amount of leading zeros on the positive RS operand. A final remainder adjust may be needed if the remainder was negative. For timing reasons, this cycle is taken even if the remainder adjust is not needed. Finally, sign adjustment is performed if needed on the quotient and/or remainder. If both operands are positive, this cycle may be skipped.

Some applications demand faster division. Many techniques may be used to increase the performance of division. For example, the Sweeney, Robertson, and Tocher (SRT) algorithm or some variation thereof may be used.

Referring to FIG. 8, multiplication operations are implemented using a finite state machine. Multiplication begins in IDLE state 8010. The multiplier stays in the idle state until the start signal is asserted. Then, the multiplier transitions to either the ARR1 state

8020 or the ARR2A state 8030 depending on whether operand RT contains a 32-bit or 16-bit value. If a 16-bit value is stored in RT, then the system transitions to state ARR2A 8030 where the first array pass is run. Then, the multiplier transitions to state ARR2B 8040 where the second array pass is run. If a 16-bit value is stored in operand RT, the multiplication is run through the array unit in state ARR1 8020.

In this implementation, the multiplier is pipelined. One multiplication may be run through the array unit and another through the CPA. Thus, the multiplier either transitions from ARR1 8020 or ARR2B 8040 to state CPA 8050 if there is no additional multiplication to perform, or begins a second multiplication. If no additional multiplication is needed, the multiplier is run through CPA 8050 and then either returns to IDLE 8010 or begins a new multiplication as discussed above.

If a second multiplication is ready to be performed when the first multiplication is ready to be run through the CPA, then the multiplier either transitions to CPA1 8060 (for a 32x16 multiplication) or CPA2A 8070 (for a 32x32 multiplication). In state CPA1 8060, the first multiplication is run through the CPA and the second multiplication is run through the array unit. The multiplier then transitions to state CPA 8050 to finalize the second multiplication.

If the second multiplication is a 32-bit multiplication, then in state CPA2A 8070 the first multiplication is run through the CPA and the second multiplication is run through the array unit. The multiplier then transitions to state ARR2B 8040 to complete the 32x32 multiplication. This pipelined approach allows 32x16 multiplications to be issued every clock cycle, with a two-cycle latency. Also, 32x32 multiplications may be issued every other clock cycle, with a three-cycle latency.

Referring to FIG. 9, iterative division operations may be implemented using a finite state machine. In one implementation, the MDU begins in IDLE state 9010. When a signal is received to begin a division operation, the MDU either transitions to DIV1 9020 if the operation is signed or DIV1U 9030 if the operation is unsigned. States DIV1 9020 and ERLY 9040 are used to prepare signed operands for division, adjusting the signs as necessary. States DIV1U 9030 and ERLYU 9050 are used to prepare an unsigned division operation. In states ERLY 9040 and ERLYU 9050, leading zeros are detected in operand RS to adjust the number of division iterations necessary.

Iterative division is performed in states DIV 9060 and DIVU 9070. Division may be performed by using a series of iterative add/subtracts and shifts. Finally, the remainders are finalized in states REM 9080 and REMU 9090. If either of the operands is negative, sign adjustment is performed in state SGN 9100.

In addition to multiplier implementations using hardware (e.g., within a microprocessor or microcontroller) implementations also may be embodied in software disposed, for example, in a computer usable (e.g., readable) medium configured to store the software (i.e., a computer readable program code). The program code causes the enablement of the functions or fabrication, or both, of the systems and techniques disclosed herein. For example, this can be accomplished through the use of general programming languages (e.g., C, C++), hardware description languages (HDL) including Verilog HDL, VHDL, AHDL (Altera HDL) and so on, or other available programming and/or circuit (i.e., schematic) capture tools. The program code can be disposed in any known computer usable medium including semiconductor, magnetic disk, optical disk (e.g., CD-ROM, DVD-ROM) and as a computer data signal embodied in a computer usable (e.g., readable) transmission medium (e.g., carrier wave or any other medium including digital, optical, or analog-based medium). As such, the code can be transmitted over communication networks including the Internet and intranets.

It is understood that the functions accomplished and/or structure provided by the systems and techniques described above can be represented in a core (e.g., a microprocessor core) that is embodied in program code and may be transformed to hardware as part of the production of integrated circuits. Also, the systems and techniques may be embodied as a combination of hardware and software. Accordingly, other implementations are within the scope of the following claim.